Portable Parallel Programming for the Dynamic Load Balancing of Unstructured Grid Applications

Rupak Biswas[1], Sajal K. Das[2], Daniel J. Harvey[2], and Leonid Oliker[3]

[1] MRJ Technology Solutions, NASA Ames Research Center, Moffett Field, CA 94035.
[2] Department of Computer Sciences, University of North Texas, Denton, TX 76203.
[3] RIACS, NASA Ames Research Center, Moffett Field, CA 94035.
{rbiswas,oliker}@nas.nasa.gov, {das,harvey}@cs.unt.edu

Abstract

The ability to dynamically adapt an unstructured grid (or mesh) is a powerful tool for solving computational problems with evolving physical features; however, an efficient parallel implementation is rather difficult, particularly from the viewpoint of portability on various multiprocessor platforms.

We address this problem by developing PLUM, an automatic portable (i.e., architecture-independent) framework for performing adaptive numerical computations in a message-passing environment. PLUM requires that all data be globally redistributed after each mesh adaptation to achieve load balance. We present an algorithm for minimizing this remapping overhead by guaranteeing an optimal processor reassignment. We also show that the data redistribution cost can be significantly reduced by applying our heuristic processor reassignment algorithm to the default mapping of the parallel partitioner. Portability is examined by comparing performance on an IBM SP2, an SGI Origin2000, and a CRAY T3E. Results show that PLUM can be successfully ported to different platforms without any code modifications.

We also present a general-purpose load balancer that utilizes symmetric broadcast networks (SBN) as the underlying communication pattern, with a goal to provide a global view of system loads across processors. Experimental results on an SP2 and Origin2000 demonstrate the portability of our novel approach which achieves superb load balance at the cost of minimal extra overhead.

Key words: Architecture-independent parallel programming, Cray 3TE, dynamic load balancing, experimental study, IBM SP2, message-passing programming, MPI, portability analysis, SGI Origin2000, unstructured mesh adaptations.

1 Introduction

The success of parallel computing in solving real-life, computation-intensive problems relies on their efficient mapping and execution on commercially available multiprocessor architectures. When the algorithms and data structures corresponding to these problems are dynamic in nature in the sense that their computational workloads grow or shrink at runtime, or when they are intrinsically unstructured, mapping them on to distributed-memory parallel machines with dynamic load balancing offers considerable challenges. Dynamic load balancing aims to balance processor workloads at runtime while attempting to minimize the communication between processors. A problem is therefore load balanced when processors have nearly equal loads with reduced communication

among themselves. With the proliferation of parallel computing, dynamic load balancing has become extremely important in several disciplines like scientific computing, task scheduling, sparse matrix computations, parallel discrete event simulation, and data mining.

The ability to dynamically adapt an unstructured mesh is a powerful tool for efficiently solving computational problems with evolving physical features. Standard fixed-mesh numerical methods can be made more cost-effective by locally refining and coarsening the mesh to capture these phenomena of interest. Unfortunately, an efficient parallelization of these adaptive methods is rather difficult, primarily due to the load imbalance created by the dynamically-changing nonuniform grid. This requires significant communication at runtime, leading to idle processors and adversely affecting the total execution time. Nonetheless, it is generally thought that unstructured adaptive-grid techniques will constitute a significant fraction of future high-performance supercomputing.

As an example, if a full-scale problem in computational fluid dynamics were to be solved efficiently in parallel, dynamic mesh adaptation would cause load imbalance among processors. This, in turn, would require large amounts of data movement at runtime. It is therefore imperative to have an efficient dynamic load balancing mechanism as part of the solution procedure. However, since the computational mesh will be frequently adapted for unsteady flows, the runtime load also has to be balanced at each step. In other words, the dynamic load balancing procedure itself must not pose a major overhead. Although several dynamic load balancers have been proposed for multiprocessor platforms [3, 4, 6, 14, 23, 32, 33], most of them are inadequate for adaptive unstructured grid applications because they lack a global view of system loads across processors. Also, workload migration in these approaches does not take into account the structure of the adaptive grid. This motivates our work.

We have developed a novel method, called PLUM, that dynamically balances processor workloads with a global view when performing adaptive numerical calculations in a parallel message-passing environment. The mesh is first partitioned and mapped among the available processors. Once an acceptable numerical solution is obtained, the mesh adaptation procedure [29] is invoked. Mesh edges are targeted for coarsening or refinement based on an error indicator computed from the solution. The old mesh is then coarsened, resulting in a smaller grid. Since edges have already been marked for refinement, the new mesh can be exactly predicted before actually performing the refinement step. Program control is thus passed to the load balancer at this time. If the current partitions will become load imbalanced after adaptation, a repartitioner is used to divide the new mesh into subgrids. The new partitions are then reassigned among the processors in a way that minimizes the cost of data movement. If the remapping cost is compensated by the computational gain that would be achieved with balanced partitions, all necessary data is appropriately redistributed. Otherwise, the new partitioning is discarded. The computational mesh is then refined and the numerical calculation is restarted. We present a heuristic algorithm to minimize the remapping overhead by guaranteeing an optimal processor reassignment, and show that the data distribution cost can be significantly reduced by applying our heuristic to the default mapping of the parallel partitioner. Examining the performance of PLUM for an actual workload (which simulates an acoustic wind-tunnel experiment of a helicopter rotor blade) on three state-of-the-art commercial parallel machines (IBM SP2, SGI Origin2000, and Cray T3E) demonstrates that PLUM can be successfully ported to different platforms without any code modifications. Preliminary versions of these results appeared in [27, 28].

We propose another new approach to dynamic load balancing for unstructured grid applications. This is based on defining a robust communication pattern (logical or physical) among processors, called *symmetric broadcast networks* (SBN), originally proposed by in [8]. Our earlier

experiments with synthetic loads [6] have demonstrated that an SBN-based load balancing solution achieves superior performance when compared to other popular techniques such as Random, Gradient, Receiver Initiated, Sender Initiated, and Adaptive Contracting. The global, dynamic load balancer developed in this paper is adaptive and decentralized in nature, and can be ported to any topological architecture through efficient embedding techniques. The SBN-based solution has been implemented on an SP2 and an Origin2000, and their performance with respect to portability has been analyzed for an adaptive unsteady grid workload which is generated by propagating a simulated shock wave within a cylindrical volume. The results show that our approach reduces the redistribution cost at the expense of a minimal extra communication overhead. In many mesh adaptation applications in which the data redistribution cost dominates the processing and communication cost, this is an acceptable trade-off. A preliminary version of this paper appeared in [7].

The results presented in this paper are also relevant in the context of the Information Power Grid (IPG) [16] which is intended to provide a convenient interface to the vast, heterogeneous, and geographically-separated computing resources of NASA and/or other IPG partners. The interface will hide details of machine particulars, such as location, size, connectivity, and name, thereby presenting a unified virtual machine. The Globus project [15] (a joint effort by Argonne National Laboratory and the University of Southern California's Information Sciences Institute) is developing a basic software infrastructure for computations that integrate geographically-distributed computational and information resources. It is, at present, the most realistic starting point for the implementation of the IPG.

There are a number of software challenges that must be addressed before actual applications can be run in a distributed fashion on the IPG. One of the most important issues will be to develop efficient architecture-independent (i.e., portable) solutions for real-life problems (e.g. a numerical solver) running on various machine platforms connected by the IPG. Another important issue is to define performance metrics for assessing the impact of high-latency communications on running a single instance of the application (solver) on multiple clusters (supernodes) of machines, given the deep levels of memory hierarchy. This will guide the design of more latency-tolerant implementations of the solver in which communication will be hidden under and/or overlapped with computation. Clearly, dynamic load balancing in such a heterogeneous environment will play a central role in the overall performance.

This paper is organized as follows. Section 2 deals with the the underlying concepts behind the dynamic load balancing of unstructured grids, and the parallel architectures and programming models used. Section 3 proposes an architecture-independent load balancer, called PLUM, and presents its performance results on three parallel machines under an actual workload. Section 4 deals with another novel load balancer based on a topology-independent communication pattern. Section 5 concludes the paper.

2 Preliminaries

This section deals with the basic concepts involved in the dynamic load balancing of adaptive unstructured grids, the parallel machine architectures that were considered for implementing the proposed methods, the parallel programming model that was used to facilitate portability of the developed software, and the actual workloads used in our experimentations.

2.1 Dynamic Load Balancing of Adaptive Unstructured Grids

Load balancing for unstructured grids essentially consists of two components: partitioning the computational mesh to balance the processor workloads, and mapping the partitions to processors that minimizes data movement. Diffusive partitioners modify existing partitions, while global partitioners generate new subgrids that need to be mapped to processors to minimize the data movement.

2.1.1 Partitioning

Graph partitioning is NP-complete; thus, research has been focused on developing efficient heuristic algorithms. Several graph partitioning algorithms have been developed over the years, particularly for static grids. Significant progress has been made in improving the partitioning heuristics as well as in generating high-quality software.

The most general approach to graph partitioning is to use generic combinatorial optimization techniques based on cost functions. Simulated annealing [22] and genetic algorithms [21] are two techniques that yield good suboptimal solutions. Both methods require properly setting a large number of parameters that makes it difficult to obtain successful partitionings efficiently; however, they are usually very effective in fine tuning existing partitions.

Clustering is another intuitive approach to graph partitioning. The greedy algorithm described in [10] grows the first partition from a given starting point. Each new partition is then begun from the boundary of the previous partition until the whole domain is decomposed. Bandwidth reduction algorithms like Reverse Cuthill-McKee (RCM) [2] also belong to this class of partitioners. The bad aspect ratio problem associated with RCM can be reduced somewhat if the scheme is used recursively, as in recursive graph bisection (RGB) [30].

Geometry-based algorithms recursively bisect the graph by exploiting its geometric properties. Recursive coordinate bisection (RCB) [30] partitions the graph based on spatial geometric coordinates, whereas recursive inertial bisection (RIB) [26] uses inertial coordinates.

A completely different class of algorithms is based on spectral methods. Recursive spectral bisection [30] is the most famous of such partitioners, and uses a strategy based on the Fiedler vector of the Laplacian of the graph. Multidimensional spectral partitioning [13] improves RSB runtimes by performing a k-way partitioning at each recursive step. Many of these geometric and spectral methods are used with the Kernighan-Lin refinement strategy [20] to improve the partitioning quality.

Multilevel algorithms reduce partitioning times considerably by contracting the graph, partitioning the coarsened graph, and then refining it to obtain a partition for the original graph. MeTiS [18] and Jostle [34] are currently the fastest multilevel schemes that are available. Both MeTiS and Jostle are available in various flavors, and can be run either as partitioners from scratch or as diffusive repartitioners.

2.1.2 Remapping

The overall effectiveness of repartitioning algorithms is determined by how successful they are in load balancing the computations while minimizing the edge-cut, as well as the cost associated with redistributing the load in order to realize the new partitioning. Data redistribution is generally considered the most expensive phase in dynamic load balancing. The migration of mesh objects requires a number of costs such as the communication overhead of remote-memory latency, and the computational overhead of rebuilding internal and shared data structures. Since global

partitioners do not consider the redistribution cost when generating subdomains, an intelligent algorithm is needed to map the new partitions onto the processors such that the data migration cost is minimized. Diffusive repartitioners explicitly attempt to minimize the data redistribution cost when generating new subdomains. These strategies are generally successful when there are gradual changes in the load distribution. However, diffusive schemes may not be well suited for problems which incur dramatic shifts in the load imbalance between redistributions. Experimental results [1] have indicated that for this class of problems, such as unsteady adaptive applications, the data migration overhead can be reduced by combining diffusive repartitioners with remapping algorithms.

Diffusion was first presented as a method for load balancing in [4]. The process can be mapped onto the diffusion equation, and much is known about its properties. In particular, it can be shown that this process will eventually converge. A nonlinear variant of this scheme which considers strip decompositions of the domain was presented in [23]. This algorithm shows better convergence properties than the standard diffusion method.

Tiling is another approach to dynamic load balancing [24]. Each processor is considered a neighborhood center, where a neighbor is defined as that processor and all others that share its subdomain boundaries. Processors within a neighborhood are balanced with respect to each other using local performance measurements. Task migration occurs from heavily-loaded to underloaded neighbors within each neighborhood. The iterative process continues until the load is globally balanced. Iterative tree balancing (ITB) [11] follows the basic tiling strategy; however, the algorithm views workload requests as forming a forest of trees rather than considering a neighborhood of processors. Each tree is then linearized, and a scan operation is used to determine the amount of data migration. ITB incorporates a more global information, and has an improved worst case load imbalance over the flat tiling model if enough iterations are permitted.

Local iterative techniques are sometimes unsuitable for dynamically balancing unsteady numerical calculations. Such applications are prone to dynamically shifting the load distribution from one adaptation phase to the next, causing small regions of the domain to suddenly incur high computational costs. Local diffusion techniques would require several iterations before global convergence, or accept an unbalanced load in exchange for faster performance. Also, by limiting task movement to nearest neighbors, several hops may be needed for a task to arrive at its final destination. Since the remapping must be frequently applied, its cost can become a significant part of the overall performance and must therefore be minimized. By moving large chunks of work units directly to their destinations, the high start-up cost of interprocessor communication can be amortized. This is the strategy that has been implemented within the PLUM framework.

2.2 Parallel Machines Considered

The dynamic load balancing methods proposed in this paper are implemented on one or more of the three state-of-the-art commercial parallel machines: an IBM SP2, an SGI Origin2000, and a Cray T3E. The SP2 belongs to the class of what is commonly known as distributed-memory multicomputers, whereas the T3E and the Origin2000 are distributed shared-memory machines. The architectural features of these machines are briefly discussed below for the sake of completeness.

2.2.1 Distributed-Memory Multicomputers

In a distributed-memory or message-passing architecture, each processor has its own local memory which only it can access directly. Since there is no common bus, remote memory is not directly

addressable and data are shared by explicitly sending and receiving messages. In this message-passing model, data transfers require operations to be performed by both the sending and the receiving processors. Such architectures are inherently scalable because they do not require tight coordination from the system for maintaining address space or data coherence. A key performance advantage of these machines is the ability for the programmer to explicitly associate specific data with processes. This allows the system to effectively use the caches and memory hierarchy on each processing unit. Unfortunately, writing message-passing code is a complex task since the user must explicitly partition the data among the local processors. Once a data structure has been distributed, all sections of the code using that data must be implemented in parallel. Thus, it is very difficult to incrementally parallelize a serial code using this paradigm.

The IBM 9076 Scalable POWERParallel SP2 System is an example of a distributed-memory parallel system. Each node consists of a six-instruction issue superscalar RISC6000 processor, running at 66.7 MHz. The nodes are connected through a high-performance switch called the Vulcan chip. The topology of the switch is an any-to-any packet switched network similar to an Omega network. An advantage of this interconnection mechanism is that the available bandwidth between any pair of communicating nodes remains constant[1] regardless of where in the topology the two nodes lie.

2.2.2 Distributed Shared-Memory/CC-NUMA Machines

A distributed shared-memory architecture combines physically distributed memory together with hardware which treats the memory as a unified, global address space. Thus, all of the physical memory is directly addressable from any processor. These systems have the advantage of efficiently supporting a shared-memory programming model, while preserving scalability. A drawback, however, is the overhead necessary for maintaining address and data coherence. To maintain a single global address space, the system must be tightly connected at either the software or the hardware level. This overhead can degrade performance with increasing number of processors.

The Cray T3E is a scalable parallel system with a shared, high-performance, globally addressable memory. The processors are RISC-based DEC chip 21164 (DEC Alpha EV5), capable of 600 MFLOPS peak performance. They are cache-based, has pipelined functional units and issues multiple instructions per cycle. The network is a high-bandwidth, low-latency bidirectional 3-D torus system, which can accommodate up to 2048 processors. The T3E uses adaptive routing to allow messages on the interconnection network to be rerouted around temporary "hot spots."

Another example of a distributed shared-memory architecture is the SGI Origin2000, the first commercially available 64-bit cache-coherent nonuniform memory access (CC-NUMA) system. Each CPU is a R10000 four-way superscalar RISC processor running at speeds of up to 195 MHz. A small high-performance switch connects two CPUs, memory, and I/O. This module, called a node, is then connected to other nodes in a hypercube fashion. An advantage of this interconnection system is that additional nodes and switches can be added to create larger systems that scale with the number of processors. The memory access time is no longer uniform as in a true shared-memory system, and varies depending on its location. Each processor in the Origin2000 has a private cache; specialized hardware maintains the image of a single shared memory as well as cache coherence.

[1]The same does not hold true for direct networks such as rings, meshes, or multidimensional toroids which require an additional performance optimization to consider the number of hops between two communicating nodes.

2.3 Message-Passing Programming Model

One of the fundamental issues in parallel computing is the choice of programming paradigms. Several paradigms have evolved over the last two decades, which include shared-memory programming, message-passing protocols, active messages, data parallel programming, multithreading, and so on. Some of of these paradigms have turned out to be architecture-independent in the sense that parallel programs, software, and tools developed with their help can be easily and efficiently ported onto various machine platforms. In this paper, we use the *message-passing programming* model.

Message passing is a popular paradigm for a wide range of parallel systems since it offers programmers direct access to communication between processes on different machines. With the explicit message-passing model, processes in a parallel application have their own private address spaces and share data via explicit messages. Several vendors have demonstrated that a message-passing library can be efficiently implemented and portable. However, because of the need for explicit communication, message passing presents a complex programming interface and is difficult to debug. Furthermore, the performance of an application depends heavily on the expertise of the programmer. Programs expressed in this manner may run on distributed-memory multiprocessors, distributed shared-memory machines, or a heterogeneous network of workstations. Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) are the two most popular standards for writing message-passing programs, and have been successfully ported onto a large number of systems including distributed-memory multicomputers and distributed shared-memory (CC-NUMA) machines.

Although we have conducted extensive experiments with both PVM and MPI, we prefer MPI since it has shown superior results on high-performance architectures. Therefore, we will report our experience with using MPI for parallel implementations of load balancing algorithms for unstructured-grid applications on three different platforms. The easy portability of our parallel programs on these machines demonstrate the versatility of our approach.

2.3.1 Message Passing Interface (MPI)

MPI is currently the most widely-used standard for writing message-passing programs, with its MPI-2 specifications released recently [25]. The main goals for designing this interface are portability, efficiency, and programmability. MPI provides the syntactic and semantic core of library routines for a distributed-memory communication environment. Providing this high-level standardization allows an efficient implementation of lower-level message-passing functions, and the possibility of hardware support by various vendors. MPI is suitable for use by general MIMD programs, as well as those written in the more restricted style of SPMD. It is a flexible communication library which can be used in a heterogeneous environment and contains bindings to both C and Fortran77. MPI-2 extends the language bindings to C++ and Fortran90.

MPI provides an interface that allows processes in a parallel program to communicate with one another. In MPI-1, processes cannot be added to or deleted from an application after it has been started. The MPI-2 process model, however, allows for the dynamic creation and termination of processes after an MPI application has started (similar to that allowed by PVM). Various communication primitives are supported. Point-to-point communications allow simple blocking and nonblocking send/receive primitives. Collective communications involve groups of processes, and allow operations such as barrier synchronizations, gather/scatters, and global reductions. MPI-2 also provides one-sided communications. This feature allows one process to specify all communication parameters for both the sending and the receiving sides. This is a powerful construct which

allows implementors to take advantage of fast communication mechanisms provided by various platforms, such as coherent or non-coherent shared memory, DMA engines, hardware-supported put/get operations, and communication coprocessors.

2.4 Real Workloads

Two sets of experiments were performed to study the effectiveness of PLUM and SBN on realistic-sized meshes. First, we examine the performance of PLUM on a steady-state, helicopter rotor test case. Three different mesh subdivision strategies are examined, based on an error indicator calculated directly from the flow solution. The second experiment evaluates the efficacy of SBN in an unsteady environment. This high-speed simulation models a shock wave moving across a uniform domain over nine levels of mesh adaptation. Results demonstrate that these two dramatically different approaches to dynamic load balancing can be successfully used for adaptive unstructured problems, in an architecture-independent fashion.

2.4.1 Workload-1 (Helicopter Rotor Test Case)

The computational mesh used to evaluate the PLUM load balancer is the one used to simulate an acoustics wind-tunnel experiment of a UH-1H helicopter rotor blade [27]. A 1/7th-scale model was tested over a range of rotational speeds. A cut-out view of the initial tetrahedral mesh, consisting of 60,968 elements, is shown in Fig.1.

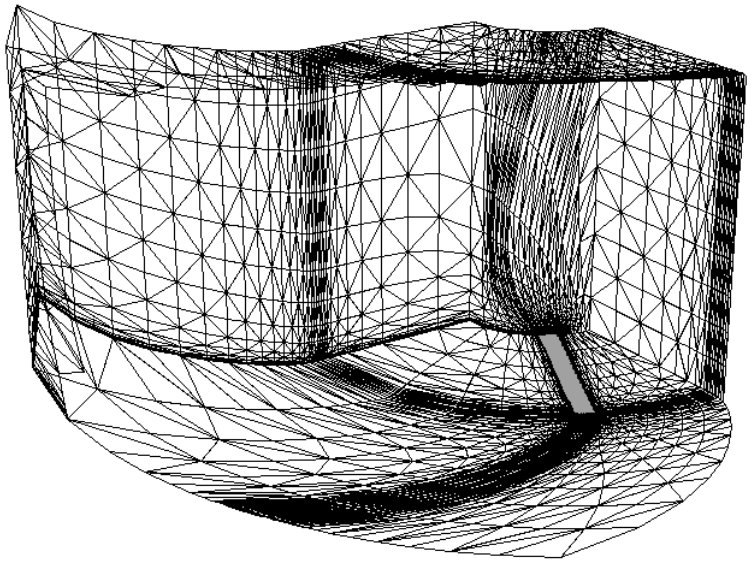

Figure 1: Cut-out view of the initial tetrahedral mesh.

Three different cases are studied, with varying fractions of the domain targeted for refinement based on an error indicator calculated directly from the flow solution. The strategies, called Real_1, Real_2, and Real_3, subdivided 5%, 33%, and 60% of the 78,343 edges of the initial mesh. This increased the number of mesh elements from 60,968 to 82,489, 201,780, and 321,841, respectively.

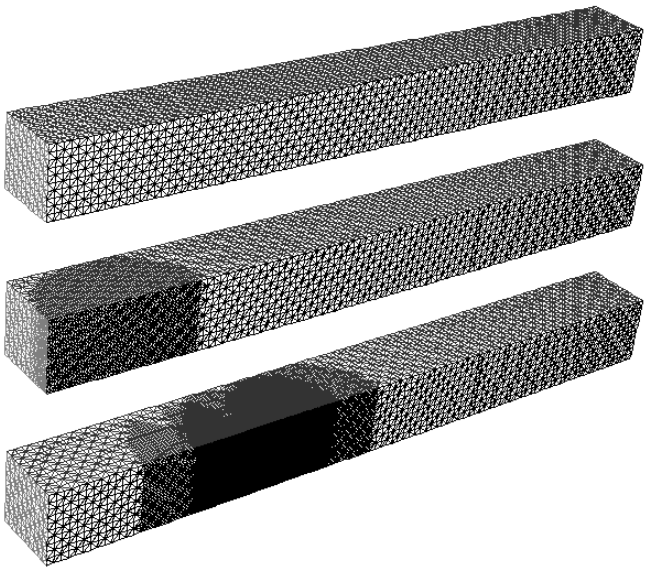

Figure 2: Initial and adapted meshes (after levels 1 and 5) for the simulated unsteady experiment.

2.4.2 Workload-2 (Unsteady Simulation Test Case)

To evaluate the SBN framework, a computational grid is used to simulate an unsteady environment where the adapted region is strongly time-dependent. This experiment is performed by propagating a simulated shock wave through the initial grid shown at the top of Fig. 2. The test case is generated by refining all elements within a cylindrical volume moving left to right across the domain with constant velocity, while coarsening previously-refined elements in its wake. Performance is measured at nine successive adaptation levels. The size of the computational mesh increased from 50,000 to 1,833,730 over the nine levels of adaptation.

3 PLUM: An Architecture-Independent Load Balancer

PLUM is an automatic and portable load balancing environment, specifically created to handle adaptive unstructured grid applications. It differs from most other similar load balancers in that it dynamically balances processor workloads with a *global* view. Prior work [1, 27] has successfully demonstrated the viability and verified the effectiveness of PLUM for various test cases involving adaptive unstructured grids. In this paper, we examine its architecture-independent feature by comparing results for test cases running on an SP2, an Origin2000, and a T3E.

Figure 3 provides an overview of PLUM. After an initial partitioning and mapping of the unstructured grid, a `Solver` executes several iterations of the application. A mesh adaptation procedure is invoked when the computational mesh is desired to be refined or coarsened. PLUM then gains control to determine if the workload among the processors has become unbalanced due to the mesh adaptation, and to take appropriate action if necessary. If load balancing is required, the adapted mesh is repartitioned and reassigned among the processors so that the cost of data movement is minimized. If the estimated remapping cost exceeds the expected computational gain to be achieved, execution continues without remapping. Otherwise, the grid is remapped among the processors before the computation is resumed.

The PLUM load balancer features (i) repeated use of the initial (mesh) dual graph during the

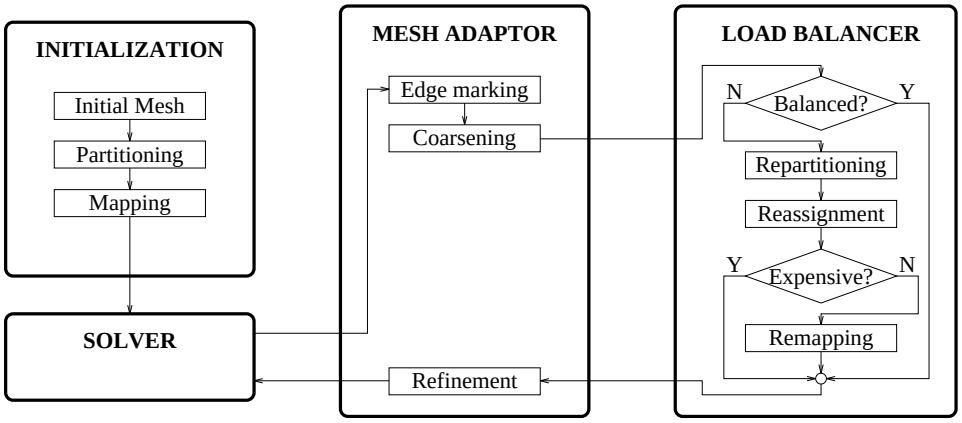

Figure 3: Overview of PLUM, an environment for globally load balancing parallel adaptive computations.

course of an adaptive computation, (ii) parallel mesh repartitioning, (iii) an efficient remapping and data movement scheme, and (iv) accurate cost model metrics. For the sake of completeness, a brief overview of these salient features are discussed in the following subsections.

3.1 Reusing the Initial Dual Graph

PLUM repeatedly utilizes the dual of the initial mesh for the purposes of load balancing. This keeps the complexity of the partitioning and reassignment phases constant during the course of an adaptive computation. New computational grids obtained by adaptation are translated by changing the *weights* of the vertices and edges of the dual graph. The weight, Wgt^v, of a dual graph vertex v models the computational workload, and is set to the number of leaf elements in the corresponding refinement tree. This is because only those elements with no children participate in the actual computation. The weight, $Remap^v$, of v models the redistribution cost, and is the total number of elements in the refinement tree. This is because all descendents of the root element must be moved from one partition to another when the load is to be rebalanced. Lastly, the weight, $Comm^e$, of a dual graph edge e models the communication cost, and is set to the number of corresponding faces in the computational mesh. The values of these three weights are used to determine an optimal partitioning that achieves balanced workloads among processors, to minimize the resulting communication, and to optimize the data movement cost.

3.2 Parallel Mesh Repartitioning

As the computational mesh is adapted and the processor workloads become unbalanced, the mesh needs to be repartitioned. PLUM can use any general-purpose partitioner that balances the computational loads and minimizes the runtime interprocessor communication. However, for PLUM to be viable, the repartitioning must be performed rapidly. Several excellent parallel partitioners are now available [11, 18, 34]; however, the results presented in this paper use PMeTiS [19].

PMeTiS is a state-of-the-art global partitioner that makes no assumptions on how the mesh is initially distributed among the processors. It is a k-way multilevel algorithm that operate in three phases: (i) in the initial coarsening phase, the original mesh, M_0, is reduced by collapsing adjacent vertices or edges through a series of smaller and smaller meshes to M_k, such that M_k

has a sufficiently small number of vertices; (ii) in the partitioning phase, the workload is balanced among the processors by using a greedy graph bisection algorithm; and (iii) in the projection phase, the partitioned mesh M_k is gradually restored to its original size M_0, while using a variant of the Kernighan-Lin refinement algorithm [20] to minimize the edge cut size.

3.3 Processor Remapping and Data Movement

The goal of processor reassignment is to find a mapping between partitions and processors that minimize the cost of redistribution. In theory, the number of new subdomains can be an integer multiple F of the number of processors. Each processor is then assigned F unique subdomains. The rational for allowing multiple subdomains per processor is that data mapping at a finer granularity reduces the volume of data movement at the cost of a slightly larger partitioning time. The first step toward processor reassignment is to compute a similarity measure that indicates how the vertex sizes of the new subdomains are distributed over the P processors. It is straightforward to represent this measure as a matrix S, where the entry S_{ij} is the sum of *Remap* values of all the dual graph vertices in the new partition j that already reside in processor i. Figure 4 shows an example of a similarity matrix for four processors where two partitions are assigned to each processor. Only the non-zero entries are shown for clarity.

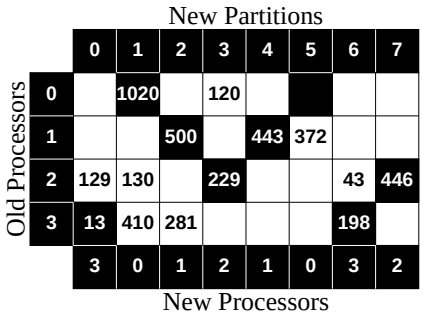

Figure 4: A similarity matrix after processor reassignment.

Various cost functions are usually needed to solve the processor reassignment problem using S for different machine architectures. We present three general metrics: `TotalV`, `MaxV`, and `MaxSR`, which model the remapping cost on most multiprocessor systems. `TotalV` minimizes the total volume of data moved among all the processors, `MaxV` minimizes the maximum flow of data to *or* from any single processor, while `MaxSR` minimizes the sum of the maximum flow of data to *and* from any processor. Experimental results [27] have indicated the usefulness of these metrics in predicting the actual remapping cost. A greedy heuristic algorithm to minimized the remapping overhead is also presented.

The `TotalV` metric assumes that by reducing network contention and the total number of elements moved, the remapping time will be reduced. Finding the optimal mapping for the `TotalV` metric can be reduced to solving the *maximally weighted bipartite graph* (MWBG) problem. The optimal algorithm has been implemented with a runtime of $O(|V|^3)$ [28], where V represents the number of partitions. The metric `MaxV`, unlike `TotalV`, considers data redistribution in terms of solving a load imbalance problem, where it is more important to minimize the workload of the most heavily-weighted processor than to minimize the sum of all the loads. We can solve for the `MaxV` metric optimally by considering the problem of finding a maximum-cardinality matching whose maximum edge cost is minimum. We refer to this as the *bottleneck maximum cardinality*

matching (BMCM) problem. The optimal BMCM algorithm has been implemented with a runtime of $O(|V|^{1/2}|E| \log |V|)$ [28], where $|E|$ is the number of entries in the similarity matrix ($|E| \approx |V|^2$). Our third metric, `MaxSR`, is similar to `MaxV` in the sense that the overhead of the bottleneck processor is minimized during the remapping phase. `MaxSR` differs, however, in that it minimizes the sum of the heaviest data flow *from* any processor and *to* any processor. This is referred to as the *double* bottleneck maximum cardinality matching (DBMCM) problem. We have developed an algorithm for computing the minimum DBMCM with a runtime of $O(|V|^{1/2}|E|^2 \log |V|)$ [28].

A greedy heuristic algorithm for solving the reassignment problem in $O(|E|)$ steps has also been developed. Oliker and Biswas [27] proved that a processor assignment obtained using the heuristic algorithm can never result in a data movement cost that is more than twice that of the optimal `TotalV` assignment. In addition, experimental results in Section 3.5 demonstrate that our heuristic quickly finds high-quality solutions for all three metrics.

3.4 Cost Model Metrics

Once the reassignment problem is solved, a model is needed to quickly predict the expected redistribution cost for a given architecture. Our redistribution algorithm consists of three major steps: first, the data objects moving out of a partition are stripped out and placed in a buffer; next, a collective communication appropriately distributes the data to its destination; and finally, the received data is integrated into each partition and the boundary information is consistently updated. Performing the remapping in this bulk fashion, as opposed to sending small individual messages, has several advantages including the amortization of message start-up costs and good cache performance.

The expected time for the redistribution procedure on bandwidth-rich systems can be expressed as $\gamma \times \texttt{MaxSR} + O$ where $\texttt{MaxSR} = \max(\texttt{ElemsSent}) + \max(\texttt{ElemsRecd})$, γ represents the total computation and communication cost to process each redistributed element, and O is the predicted sum of all constant overheads [27]. This formulation demonstrates the need to model the `MaxSR` metric when performing processor reassignment. By minimizing `MaxSR`, we can guarantee a reduction in the computational overhead of our remapping algorithm. Since the computational workload is architecture independent, we are effectively solving two load balancing problems separated by a collective communication. Moreover, by reducing `MaxSR`, we can achieve a savings in the communication overhead on many bandwidth-rich systems. In order to compute the values of γ and O, a simple least squares fit through several data points for various redistribution patterns and their corresponding runtimes can be used. This procedure needs to be performed only once for each architecture, and the values of γ and O can then be used in actual computations to estimate the redistribution cost.

3.5 Comparison of Reassignment Algorithms

Table 1 presents a comparison of our five different processor reassignment algorithms in terms of the reassignment time (in secs) and the amount of data movement. Results are shown for the Real_2 strategy with Workload-1 on the SP2 with $F = 1$. The PMeTiS [19] case does not require any explicit processor reassignment since we choose the default partition-to-processor mapping given by the partitioner. The poor performance for all three metrics is expected since PMeTiS is a global partitioner that does not attempt to minimize the remapping overhead. Previous work [1] compared the performance of PMeTiS with other partitioners.

Table 1: Comparison of reassignment algorithms for Real_2 on the SP2 with $F = 1$

	P = 32				P = 64			
Algorithm	TotalV Metric	MaxV Metric	MaxSR Metric	Reass. Time	TotalV Metric	MaxV Metric	MaxSR Metric	Reass. Time
PMeTiS	58297	5067	7467	0.0000	67439	2667	4452	0.0000
MWBG	34738	4410	5822	0.0177	38059	2261	3142	0.0650
BMCM	49611	4410	5944	0.0323	52837	2261	3282	0.1327
DBMCM	50270	4414	5733	0.0921	54896	2261	3121	1.2515
Heuristic	35032	4410	5809	0.0017	38283	2261	3123	0.0088

The execution times of the other four algorithms increase with the number of processors because of the growth in the size of the similarity matrix; however, the heuristic time for 64 processors is still very small and acceptable. The total volume of data movement is obviously the smallest for the MWBG algorithm since it optimally solves for the TotalV metric. In the optimal BMCM method, the maximum of the number of elements sent or received is explicitly minimized, but all the other algorithms give almost identical results for the MaxV metric. In our helicopter rotor experiment, only a few localized regions of the domain incur a dramatic increase in the number of grid points between refinement levels. These newly-refined regions must shift a large number of elements onto other processors in order to achieve a balanced load distribution. Therefore, a similar MaxV solution should be obtained by any reasonable reassignment algorithm.

The DBMCM algorithm optimally reduces MaxSR, but achieves no more than a 5% improvement over the other algorithms. Nonetheless, since we believe that the MaxSR metric can closely approximate the remapping cost on many architectures, computing its optimal solution can provide useful information. Notice that the minimum TotalV increases slightly as P grows from 32 to 64, while MaxSR is dramatically reduced by over 45%. This trend continues as the number of processors increases, and indicates that PLUM will remain viable on a large number of processors, since the per processor workload decreases as P increases.

Finally, observe that the heuristic algorithm does an excellent job in minimizing all three cost metrics, in a trivial amount of time. Although theoretical bounds have only been established for the TotalV metric, empirical evidence indicates that the heuristic algorithm closely approximates both MaxV and MaxSR. Similar results were obtained for the other edge-marking strategies.

3.6 Portability Analysis

The top three plots in Fig. 5 illustrate parallel speedup for the three edge-marking strategies on the SP2, Origin2000, and T3E. Two sets of results are presented for each machine: one when data remapping is performed after mesh refinement, and the other when remapping is done before refinement. The speedup numbers are almost identical on all three machines. The Real_3 case shows the best speedup values because it is the most computation intensive. Remapping data before refinement has the largest relative effect for Real_1, because it has the smallest refinement region and predictively load balancing the refined mesh returns the biggest benefit. The best results are for Real_3 with remapping before refinement, showing an efficiency greater than 87% on 32 processors.

To compare the performance on the SP2, Origin2000, and T3E more critically, one needs to look at the actual times rather than the speedup values. Table 2 shows how the execution time

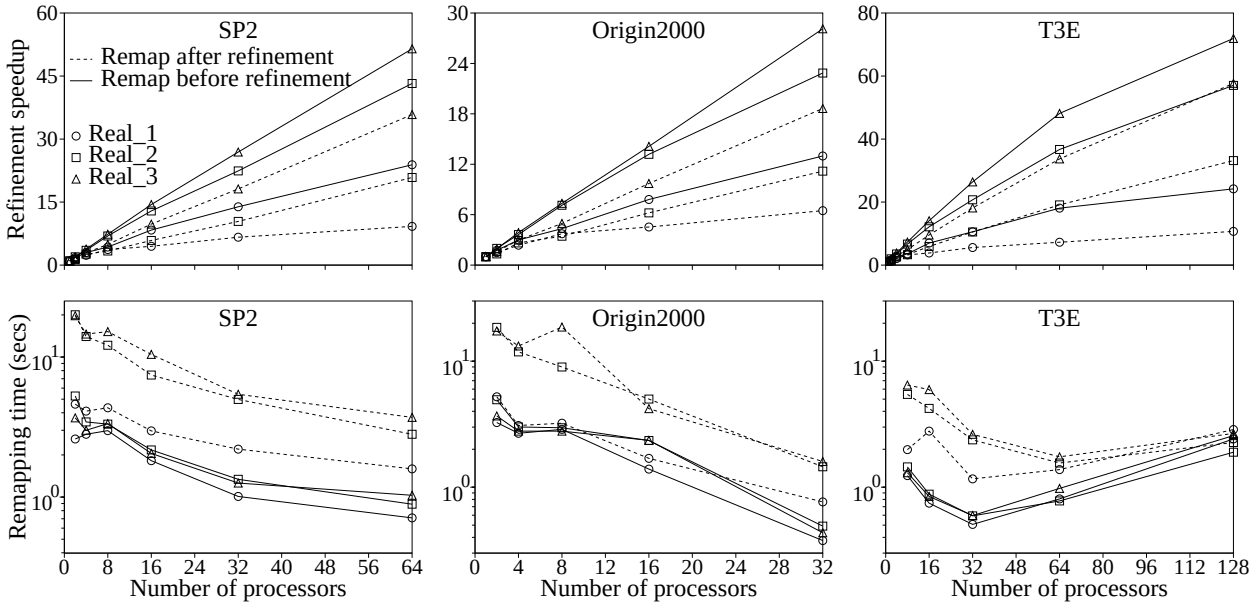

Figure 5: Refinement speedup (top) and remapping time (bottom) within PLUM on the SP2, Origin2000, and T3E, when data is redistributed after or before mesh refinement.

(in secs) is spent during the refinement and subsequent load balancing phases for the Real_2 case when data is remapped before the subdivision phase. The processor reassignment times are not presented since they are negligible compared to other times, as is evident from Table 1. Notice that the T3E adaptation times are consistently more than 1.4 times faster than the Origin2000 and three times faster than the SP2. One reason for this performance difference is the disparity in the clock speeds of the three machines. Another reason is that the mesh adaptation code does not use the floating-point units on the SP2, thereby adversely affecting its overall performance.

The bottom three plots in Fig. 5 show the remapping time for each of the three cases on the SP2, Origin2000, and T3E. In almost every case, a significant reduction in remapping time is observed when the adapted mesh is load balanced by performing data movement prior to refinement. This is because the mesh grows in size only after the data has been redistributed. In general, the remapping times also decrease as the number of processors is increased. This is because even though the total volume of data movement increases with the number of processors, there are actually more processors to share the work. The remapping times when data is moved before mesh refinement are reproduced for the Real_2 case in Table 2 since the exact values are difficult to read off the log-scale.

Perhaps the most remarkable feature of these results is the peculiar behavior of the T3E when $P \geq 64$. When using up to 32 processors, the remapping performance of the T3E is very similar to that of the SP2 and Origin2000. It closely follows the redistribution cost model given in Sec. 3.4, and achieves a significant runtime improvement when remapping is performed prior to refinement. However, for 64 and 128 processors, the remapping overhead on the T3E begins to increase and violates our cost model. The runtime difference when data is remapped before and after refinement is dramatically diminished; in fact, all the remapping times begin to converge to a single value! This indicates that the remapping time is no longer affected only by the volume of data redistributed but also by the interprocessor communication pattern. One way of potentially improving these results is to take advantage of the T3E's ability to efficiently perform one-sided communication.

Table 2: Anatomy of execution times for Real_2 on the Origin2000, SP2, and T3E

	Adaptation Time			Remapping Time			Partitioning Time		
P	O2000	SP2	T3E	O2000	SP2	T3E	O2000	SP2	T3E
2	5.261	12.06	3.455	3.005	3.440	2.648	0.628	0.815	0.701
4	2.880	6.734	1.956	3.005	3.440	1.501	0.584	0.537	0.477
8	1.470	3.434	1.034	2.963	3.321	1.449	0.522	0.424	0.359
16	0.794	1.846	0.568	2.346	2.173	0.880	0.396	0.377	0.301
32	0.458	1.061	0.333	0.491	1.338	0.592	0.389	0.429	0.302
64		0.550	0.188		0.890	0.778		0.574	0.425
128			0.121			1.894			0.599

Another surprising result is the dramatic reduction in remapping times when using 32 processors on the Origin2000. This is probably because network contention with other jobs is essentially removed when using the entire machine. When using up to 16 processors, the remapping times on the SP2 and the Origin2000 are comparable, while the T3E is about twice as fast. Recall that the remapping phase within PLUM consists of both communication and computation. Since the results in Table 2 indicate that computation is faster on the Origin2000, it is reasonable to infer that bulk communication is faster on the SP2. These results generally demonstrate that our methodology within PLUM is effective in significantly reducing the data remapping time and improving the parallel performance of mesh refinement.

Table 2 also presents the PMeTiS partitioning times for Real_2 on all three systems; the results for Real_1 and Real_3 are almost identical because the time to repartition mostly depends on the initial problem size. There is, however, some dependence on the number of processors used. When there are too few processors, repartitioning takes more time because each processor has a bigger share of the total work. When there are too many processors, an increase in the communication cost slows down the repartitioner. Table 2 demonstrates that PMeTiS is fast enough to be effectively used within our framework, and that PLUM can be successfully ported to different platforms without any code modifications.

4 A Topology-Independent Load Balancer

In this section, we propose a load balancer based on a *symmetric broadcast network* (SBN), which takes into account the global view of system loads among the processors. The SBN is a robust, topology-independent communication pattern (logical or physical) among P processors in a multicomputer system [8]. Before applying it to load balancing for adaptive grids, let us give a brief overview of SBN.

4.1 Symmetric Broadcast Networks

An SBN of dimension $d \geq 0$, denoted as SBN(d), is a $(d+1)$-stage interconnection network with $P = 2^d$ processors in each stage. It is constructed recursively as follows. A single node forms the basis network SBN(0). For $d > 0$, an SBN(d) is obtained from a pair of SBN($d-1$)s by adding a communication stage in the front with the following additional interprocessor connections:

- Node i in stage 0, is made adjacent to node $j = (i + P/2) \bmod P$ of stage 1; and

- Node j in stage 1 is made adjacent to the node in stage 2 which was the stage 0 successor of node i in SBN$(d-1)$.

Figure 6 depicts an SBN(2), recursively constructed from two SBN(1)s.

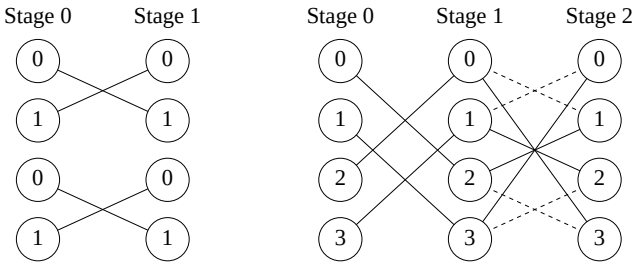

Figure 6: Construction of SBN(2) from a pair of SBN(1)s. The new connections are shown by solid lines and the original connections by dashed lines.

By definition, each node in every stage s, for $1 \leq s \leq d-1$, of SBN(d) has exactly two successors; whereas a node in stage 0 has only one successor and a node in stage d has no successor. The SBN(d) defines unique communication patterns (or broadcast trees) among the nodes in the network. Precisely, for any source node or root x at stage 0, where $0 \leq x < P$, there exists a unique broadcast tree T_x of height $d = \log P$ such that each of the 2^d nodes appears exactly once.

Let n_x^s be a node at stage s in the broadcast tree T_x having the root node x (at stage 0), where $0 \leq x < P$. Then $n_x^s = n_0^s \oplus x$, where $\oplus$ is the exclusive-OR operator, thus leading to $T_x = T_0 \oplus x$. Thus, all SBN communication patterns can be derived from the template tree with node 0 as the root. Furthermore, the predecessor and successors of each node can be uniquely defined by specifying the root x and the communication stage s so that messages from x can be appropriately routed to the other nodes. As shown in [5], the versatility of the SBN lies in its efficient embeddings into other topologies such as hypercubes and meshes.

4.2 SBN-Based Load Balancing

The proposed SBN-based load balancer can be classified as adaptive and decentralized. It takes into account a global view of the system and makes it effective for adaptive grid applications. The SBN load balancer processes two types of messages: (i) load balance messages and (ii) job distribution messages.

The first type of messages is broadcast when a processor p determines that its weighted queue length `QWgt`(p) < `MinTh`, a minimum threshold. (In our experiments, `MinTh`, was set to reflect a 0.1 second processing load.) A load balancing message will also be broadcast if `QWgt`(p) > `MaxTh`, a maximum threshold, or if distributing the excess jobs will result in other processors exceeding `MaxTh`. As the load balancing message passes from one node to another, values for the global weighted queue length (`GWLen`) and the weighted system load level (`WSysLL`) are computed.

The second type of messages are used to distribute jobs when `QWgt`(p) > `MaxTh`. Distribution messages are also used to complete the load balancing process. After the `WSysLL` value is calculated, a distribution message is broadcast through the network when jobs are routed to the lightly-loaded processors and the threshold values (`MinTh` and `MaxTh`) are updated. As a result, the workload at all nodes is balanced.

Let us now discuss the various parameters and other details involved in the load balancer implementation, a preliminary version of which appeared in [7].

- **Weighted Queue Length:** The queue length of a processor p by itself is an accurate estimate of the amount of time required to completely service the vertices in its local queue, particularly in applications where the grid is adapted. To achieve a better balance, we must take into account the system variables like computation, communication, and redistribution costs, that affect the processing of a local queue. Therefore, we define a metric called weighted queue length, $\texttt{QWgt}(p)$. Let Wgt^v be the computational cost to process a vertex v, $Comm_p^v$ be the communication cost to interact with the vertices adjacent to v but whose data sets are not local to p, and $Remap_p^v$ be the redistribution cost to copy the data set for v to p from another processor. These factors vary widely from one vertex to another in a given grid. $\texttt{QWgt}(p)$ is defined as:

$$\texttt{QWgt}(p) = \sum_{v \text{ assigned to } p} (Wgt^v + Comm_p^v + Remap_p^v).$$

  Clearly, if the data set for v is already assigned to p, no redistribution cost is incurred, i.e., $Remap_p^v = 0$. Similarly, if the data sets of all the vertices adjacent to v are also already assigned to p, the communication cost, $Comm_p^v$, is 0.

- **Weighted System Load:** The weighted system load is defined as:

$$\texttt{WSysLL} = \left\lceil \frac{1}{P} \sum_{i=1}^{P} \texttt{QWgt}(i) \right\rceil,$$

  where P is the total number of processors used. Assuming that the load is perfectly balanced among the processors, `WSysLL` estimates the time required to process the grid and reflects the processing, communication, and redistribution costs in the current grid-to-processor assignment. Hence, a global view of the system is captured.

- **Prioritized Vertex Selection:** When selecting vertices to be processed, the SBN load balancer takes advantage of the underlying structure of the adaptive grid and defers local execution of boundary vertices as long as possible because they may be migrated for more efficient execution. Thus, the next queued vertex is selected such that it minimizes the overall cut size of the adapted grid. A priority min-queue is maintained for this purpose, where the priority of a vertex v in processor p is given by $(Comm_p^v + Remap_p^v)/Wgt^v$. Therefore, vertices with no communication and redistribution costs are executed first. Those with low communication or redistribution overhead relative to their computational weight are processed next. Conceptually, internal vertices are processed before those on partition boundaries.

- **Differential Edge Cut:** For balancing the system load among processors, an optimal policy for vertex migration needs to be established. When vertices are being moved, assume that processor p is about to reassign some of its vertices to another processor q. The SBN load balancer running on p randomly picks a subset of vertices from those queued locally. For each selected vertex v, the differential edge cut[2], $\Delta\texttt{Cut}$, is calculated as follows:

$$\Delta\texttt{Cut} = Remap_q^v - Remap_p^v + Comm_q^v - Comm_p^v.$$

[2]Here we are deviating from the usual definition of edge cut to account for the dynamic nature of the SBN load balancer.

If Δ`Cut` > 0, it is normalized as Δ`Cut`$/Wgt^v$. A positive Δ`Cut` indicates that an increase in communication and redistribution costs will result if v is migrated from p to q. Therefore, the formula favors migrating vertices with the smallest increase in communication cost per unit computational weight. In contrast, negative Δ`Cut` values indicate a reduction in communication and redistribution costs, hence favoring the migration of vertices with the largest absolute reduction in communication and redistribution costs.

Once Δ`Cut` is calculated for all the randomly chosen vertices, the vertex `MinV` with the smallest value of Δ`Cut` is chosen for migration. Next, following a breadth-first search, the SBN balancer selects the vertices adjacent to `MinV` that are also queued locally for processing at p. The breadth-first search stops either when no adjacent vertices are queued for local processing at p, or if a sufficient number of vertices have been found for migration. If more vertices still need to be migrated, another subset of vertices are randomly chosen and the procedure is repeated. This migration policy strives to maintain or improve the cut size during the execution of the load balancing algorithm. In contrast, tradidional dynamic load balancing algorithms do not consider cut size and hence are likely to experience larger cut sizes as execution proceeds.

- **Data Redistribution Policy:** The redistribution of data is performed in a "lazy" manner. Namely, the data set for a given vertex v in a processor p is not moved to q until the latter processor is about to execute v. Furthermore, the data sets of all vertices adjacent to v that are assigned to q are migrated as well. This policy greatly reduces both the redistribution and communication costs by avoiding multiple migrations of data sets and having resident all adjacent vertices that are assigned to processor q while v is being processed.

  Data migration is implemented by broadcasting a job migration message when a vertex is about to be processed and its corresponding data set is not resident on the local processor. A locate-message is then broadcast to indicate the new location of the data set. This policy is expected to maximize the number of adjacent vertices that are local when a grid point is processed. Hence, by considering the underlying grid structure, the communication overhead is reduced.

4.3 Differences with PLUM

The SBN-based load balancer differs from PLUM in several ways:

- Processing is temporarily halted under PLUM while the load is being balanced. During the suspension, a new partitioning is generated and data is redistributed among the nodes of the network. The SBN approach, on the other hand, allows processing to continue while the load is dynamically balanced. This feature also allows for the possibility of utilizing latency-tolerant techniques to hide the communication and redistribution costs during processing.

- Under PLUM, suspension of processing and subsequent repartitioning does not guarantee an improvement in the quality of load balance. If it is determined that the estimated remapping cost exceeds the expected computational gain that is to be achieved by a load balancing operation, processing continues using the original grid assignment. This could result in unnecessary idle time. In contrast, the SBN approach will always result in improved load balance among processors.

- PLUM redistributes all necessary data to the appropriate nodes immediately before processing continues. SBN, however, distributes in a "lazy" manner. Data is migrated to a processor only when it is ready to process the data, thus reducing redistribution and communication overhead.

- The load balancing under PLUM takes place *before* the solver phase of the computation, whereas SBN balances the load *during* the solver execution. We therefore cannot directly compare PLUM and SBN, since their relative performance is solver dependent. Future work will integrate SBN into the adaptation phase making a direct comparison possible.

4.4 Performance Metrics

The following metrics are chosen to evaluate the effectiveness of the SBN load balancer when processing an unsteady adaptive grid. Recall that v denotes a vertex to be processed and P is the number of processors.

- **Maximum Redistribution Cost:** The goal is to capture the total cost of packing and unpacking data, separated by a barrier synchronization. Since a processor can either be sending or receiving data, the overhead of these two phases is modeled as a sum of two costs in this metric:

$$\texttt{MaxSR} = \max_{p \in P} \Big\{ \sum_{v \text{ sent from } p} Remap_p^v \Big\} + \max_{p \in P} \Big\{ \sum_{v \text{ recv by } p} Remap_p^v \Big\}.$$

  Since MaxSR pertains to the processor that incurs the maximum redistribution cost, a reduction in the total data redistribution overhead can be guaranteed by minimizing MaxSR.

- **Load Imbalance Factor:** It is formulated as:

$$\texttt{LoadImb} = \max_{p \in P} \texttt{QWgt}(p) \;/\; \texttt{WSysLL}.$$

  This factor should be as close to unity as possible.

- **Cut Percentage:** The runtime interaction between adjacent vertices residing on different processors is represented by this metric as:

$$\texttt{Cut\%} = 100 \times \sum_{p \in P} \sum_{v \text{ assigned to } p} Comm_p^v \Big/ \sum_{e \text{ in mesh}} Comm^e \,,$$

  where $Comm^e$ is the weight of edge e in the adaptive grid. The Cut% value should be as small as possible.

  Pre-Exec Cut% in Table 3 initially projects the grid edge cut before processing an adaptation level but after the previous adaptation level has been processed.

  Post-Exec Cut% is the actual cut realized after processing a given adaptation level.

Table 3: Grid adaptation results on SP2 using SBN-based balancer

Adaptation Level	Pre-Exec Cut %	Post-Exec Cut %	MaxSR	LoadImb
P=2				
1	0.09%	4.64%	6,974	1.00
2	3.14%	6.18%	30,538	1.00
3	5.36%	6.08%	57,724	1.00
4	3.93%	3.86%	20,646	1.00
5	2.91%	5.32%	76,893	1.00
6	2.33%	4.62%	103,544	1.00
7	2.23%	5.86%	140,904	1.00
8	2.83%	6.14%	153,735	1.00
9	3.10%	6.89%	129,374	1.00
Avg	3.14%	5.54%	80,037	1.00
P=4				
1	2.26%	8.15%	4,078	1.00
2	7.22%	10.01%	26,187	1.00
3	9.44%	11.69%	64,110	1.00
4	9.16%	9.48%	46,406	1.00
5	6.60%	11.86%	149,042	1.00
6	9.83%	10.89%	94,269	1.00
7	6.58%	8.00%	50,337	1.00
8	2.79%	15.31%	170,408	1.00
9	11.53%	11.48%	85,152	1.00
Avg	7.86%	11.17%	76,665	1.00
P=8				
1	6.66%	10.77%	2,518	1.01
2	13.93%	14.98%	11,109	1.00
3	15.11%	18.16%	46,088	1.00
4	14.65%	15.83%	53,032	1.00
5	11.09%	16.48%	69,583	1.00
6	11.02%	15.91%	85,982	1.00
7	13.75%	18.13%	105,946	1.00
8	12.84%	19.51%	28,974	1.00
9	15.34%	17.35%	80,477	1.00
Avg	13.30%	17.18%	53,745	1.00
P=16				
1	15.36%	20.61%	1,767	1.01
2	24.82%	25.56%	7,259	1.00
3	24.40%	27.45%	36,031	1.01
4	20.60%	22.77%	43,943	1.01
5	16.11%	24.27%	71,736	1.01
6	17.83%	22.28%	66,211	1.01
7	19.75%	25.00%	55,361	1.01
8	17.83%	25.30%	64,796	1.01
9	17.87%	21.59%	74,316	1.01
Avg	19.19%	24.02%	46,825	1.01
P=32				
1	21.59%	26.74%	1,184	1.01
2	30.35%	32.32%	4,387	1.02
3	30.06%	34.04%	8,445	1.02
4	27.28%	31.43%	41,783	1.01
5	21.35%	29.40%	42,843	1.01
6	24.04%	29.42%	42,688	1.01
7	22.35%	30.45%	41,347	1.02
8	20.59%	30.48%	37,006	1.02
9	22.19%	29.43%	32,594	1.02
Avg	23.97%	30.58%	28,031	1.02

4.4.1 Experimental Results on SP2

The SBN-based load balancing algorithm was first implemented using MPI on the wide-node IBM SP2 located at NASA Ames Research Center, and tested with Workload-2 obtained from adaptive calculations of time-dependent, unsteady simulated shock wave. Table 3 presents the performance results. In addition to achieving excellent load balance, the redistribution cost is significantly reduced. However, the edge cut percentages are somewhat higher, indicating that the SBN strategy reduces the redistribution cost at the expense of a slightly higher communication cost.

For example, `LoadImb` = 1.02 for $P = 32$. When $P \leq 8$, an ideal load imbalance factor of 1.00 is achieved for most of the adaptation levels. The `MaxSR` metric indicates the amount of redistribution cost incurred while processing the adaptive grid. For $P = 32$, we obtained `MaxSR` = 28,031. This is due to the SBN "lazy" approach to migration of vertex data sets.

Table 4: Communication overhead of the SBN load balancer.

P	Balancing Messages (bytes)	Balancing Bandwidth (%)	Migration Messages (bytes)	Migration Bandwidth (%)
2	342,456	0.00	3,918,912	3.67
4	149,964	0.00	7,939,344	7.44
8	463,340	0.01	25,397,376	23.79
16	581,432	0.02	30,453,888	28.53
32	1,550,292	0.12	38,244,384	35.83

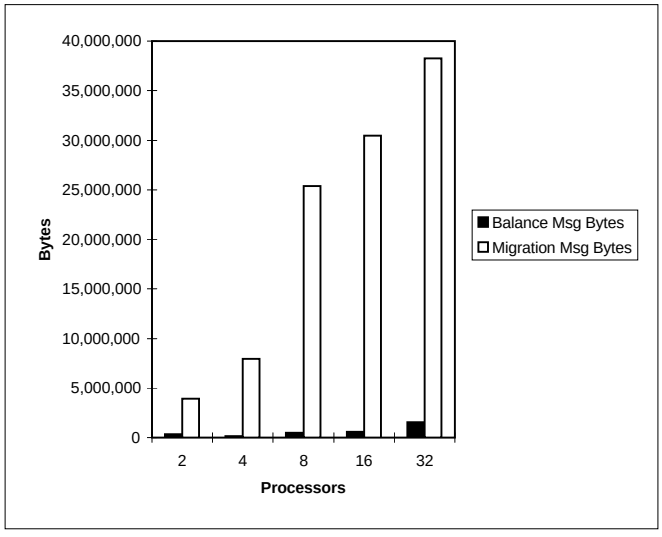

Figure 7: SBN load balancing related communication overhead.

Tables 4, 5 and Figs 7, 8 show measurements of overheads due to message passing and processing according to experiments run on the SP2. Table 4 gives the number of bytes that were transferred between processors during the load balancing and the job distribution phases. The number of bytes transferred is also expressed as a percentage of the available bandwidth. A wide-node SP2 has a message bandwidth of 36 megabytes/second and a message latency of 40 micro seconds. Figure 7

Table 5: Percentage overhead of the SBN load balancer.

P	Balancing Activity	Migration Activity	Vertex Selection
2	0.0053	0.0014	0.4530
4	0.0087	0.0069	2.0381
8	0.1745	0.0569	2.8386
16	0.2669	0.0629	0.8845
32	0.1154	0.0774	2.1043

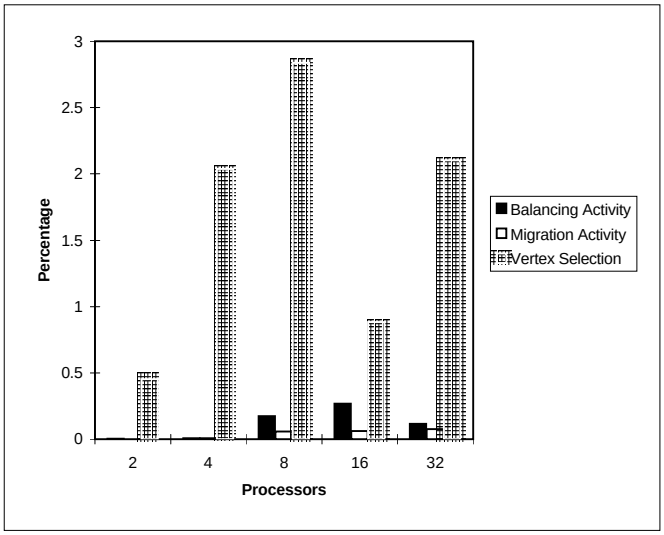

Figure 8: SBN load balancing related processing overhead.

plots this message passing overhead graphically, and demonstrates that the cost of vertex migration is significantly greater than the cost of actually balancing the system load. This is not unexpected. An extrapolation of the results using an exponential curve-fitting program indicates that normal speedup will not scale past 128 processors. The formula derived by the curve-fitting program for total overhead, T_{ovhd}, due to processing and message passing is obtained as:

$$T_{ovhd} = 12215.8776 \times P^{\log 0.5777} + 2.4870 \times P^{\log 1.8041} + 0.1023 \times P^{\log 1.5451}.$$

As expected, much of the overhead is due to the latency associated with transmitting many small messages which is represented by the dominating term, $P^{\log 1.8041}$. However, since this exponent is less than one, the overhead is asymptotically sublinear.

Table 5 shows the fraction of time spent in the SBN load balancer compared to the execution time required to process the mesh adaptation application. The three columns correspond to three categories of load balancing activities: (i) the time needed to handle balance related messages, (ii) the time needed to migrate messages from one processor to another, and (iii) the time needed to select the next vertex to be processed. Figure 8 shows graphical plots. The results show that processing related to the selection of vertices is the most expensive phase of the SBN load balancer. However, the total time required to load balance is still relatively small compared to the time spent processing the mesh.

In conclusion, these experimental results demonstrate that the proposed SBN-based dynamic load balancer is effective in processing adaptive grid applications, thus providing a global view across processors. In many grid applications in which the cost of data redistribution dominates the cost of communication and processing, the SBN balancer would be preferred.

4.4.2 Results on Origin2000

We were able to directly port the SBN methodology from the SP2 to the Origin2000 machine without any code modifications. (The only modifications, as discussed below, were made either to refine our approach or to take advantage of the distributed-shared memory architecture of Origin2000.) This demonstrates the architecture independence of our load balancer.

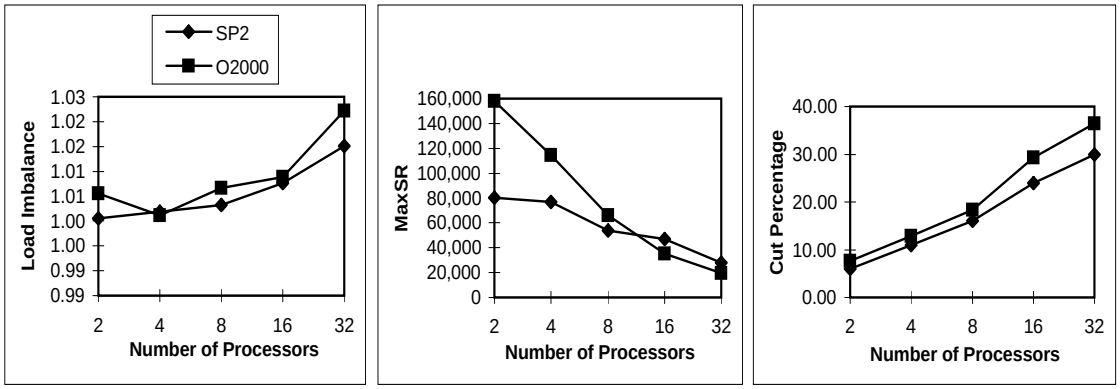

Figure 9: `LoadImb`, `MaxSR`, and `Cut%` on the SP2 and Origin2000.

The plots in Fig. 9 illustrate the effect upon `Cut%`, `MaxSR`, and `LoadImb` when the experiments were run on the Origin2000 as compared with those parameters for SP2. `LoadImb` percentages are almost identical with those on SP2. `Cut%` values are consistently larger on the Origin2000. But `MaxSR` values are larger on the Origin2000 when $P < 16$.

Some of the differences in performance results on these two machines are due to additional refinements that were implemented prior to running the experiments on the Origin2000. The refinements are summarized below:

- `QWgt`(p) was revised to include the cost of migrating the data sets that are assigned to p but whose corresponding vertices are to be processed elsewhere. For example, if vertex v is to be processed by processor q but its data set is resident in p, $Remap_q^v$, is added to `QWgt`(p). This refinement more accurately models processor load than what was reflected in the `QWgt`(p) definition presented in Section 4.2. Note that this modification increases the anticipated cost of vertex migration. As a result, the SBN-based balancing algorithm tolerates a greater cut size.

- The algorithm for distributing jobs was modified somewhat for the Origin2000 experiments. The purpose of this refinement is to guarantee that the processor initiating load balancing

always gets sufficient load to process, thereby reducing the total number of balancing related messages that need to be processed. A side effect of this refinement is to increase the number of vertices migrated, especially when the value of P is small.

- Another refinement was made to migrate data sets corresponding to groups of vertices at a time. This allows for fewer total migration messages. In our experiments the volume of migration messages were reduced by more than 80% as a result of this change.

We are currently running experiments on Origin2000 to measure the overheads due to balancing and migration messages, which will be reported in the final version.

5 Conclusions and Discussions

We have demonstrated the portability of our novel approaches to solving the load balancing problem for adaptive unstructured grids on three state-of-the-art commercial machines which belong to the class of distributed-memory multicomputers or distributed shared-memory machines. The experiments were conducted using actual workloads of both steady and unsteady grids obtained from real-life applications.

We are currently examining the portability of our software on other platforms (such as shared-memory environments or networks of workstations) as well as employing various parallel programming models[3]. For example, we are currently developing a shared-memory implementation for the Origin2000. Although the shared-memory paradigm is convenient and relatively easy to use, the performance depends largely on the efficiency of the underlying mechanism in accessing the shared memory and managing shared data between different processors. Fortunately, MPI could be made to work in a shared-memory environment.

Commercial supercomputer systems are designed with specialized high-speed networks. In recent years, workstations have been linked with conventional networks, such as Ethernet, FDDI, or ATM to form parallel systems. These networks (or clusters) of workstations (NOWs) are equivalent to NUMA distributed-memory parallel machines. The availability of relatively low-cost high-speed networks make this approach a good price/performance value for compute-intensive problems. However, NOWs can efficiently support fewer parallel applications than specialized MPP's due to their lower bandwidth and higher communication latency.

As regards to other programming models, Active Messages [9] offers an asynchronous communication mechanism designed to mitigate the message-passing overhead. The idea is to overlap communication latency with computation by exposing the full hardware capabilities to the programmer. Each Active Message contains a handler address in addition to the data being transferred. When messages arrive at their destination, the ongoing computation is interrupted and the handler is executed. The handler quickly extracts the message out of the network, and integrates it into a user-specified memory address with a small amount of work. This mechanism relies on a uniform code image in all communicating nodes, as is commonly used in the SPMD programming model. Active Message are intended to serve as building blocks for creating higher-level communication libraries. The MPI-2 standard, for example, allows for one-sided communication, and could be built on top of Active Messages. The disadvantage of this asynchronous communication style is increased programming complexity and higher probability of deadlock within the network. We

[3]Note that the data parallel programming paradigm is more effective for specific applications whose data can be partitioned in a regular fashion, and thus not unsuitable for irregular or adaptive computations like ours.

believe that the performance of PLUM can be improved by exploiting one-sided communications within the programs. This will be the subject of future research.

Reducing the gap between processor speed and memory latency is key for obtaining high performance on distributed-memory multiprocessors. Multithreading aims at tolerating remote memory latency by overlapping communication with computation through context switching. Threads are similar in concept to processes, but whereas a process is normally created for a completely separate application, threads run within the application itself. While operating systems normally manage processes so that one process cannot access the memory assigned to another process, a thread shares memory with all other threads in the application. It is therefore much easier for threads to communicate with each other than it is for processes. Communication latencies arising from remote memory access or cache misses can therefore be absorbed by the fast context switching of the threads. The drawback to multithreaded programming include an increase in programming complexity, a loss in portability, and the overhead necessary for thread management. Experimental multithreaded architectures such as EARTH [17] and the EM-X [31], are designed with specialized hardware support for efficiently handling thread management. We intend to also implement a mulithreaded version of our load balancing solutions.

Acknowledgements

This work is supported by NASA under Contract Numbers NAS 2-14303 with MRJ Technology Solutions and NAS 2-96027 with Universities Space Research Association, and by Texas Advanced Research Program Grant Number TARP-97-003594-013.

References

[1] R. Biswas and L. Oliker, "Experiments with repartitioning and load balancing adaptive meshes," Technical Report NAS-97-021, NASA Ames Research Center, Moffett Field, 1997.

[2] W. Chan and A. George, "A linear time implementation of the reverse Cuthill-McKee algorithm," *BIT*, 20 (1980), pp. 8–14.

[3] N. Chrisochoides, "Multithreaded model for the dynamic load balancing of parallel adaptive PDE computations," *Applied Numerical Mathematics*, 20 (1996), pp. 321–336.

[4] G. Cybenko, "Dynamic load balancing for distributed-memory multiprocessors," *Journal of Parallel and Distributed Computing*, 7 (1989), pp. 279–301.

[5] S.K. Das and D.J. Harvey, "Performance analysis of an adaptive symmetric broadcast load balancing algorithm on the hypercube," Technical Report CRPDC-95-1, Dept. of Computer Science, Univ. of North Texas, Denton, 1995.

[6] S.K. Das, D.J. Harvey, and R. Biswas, "Adaptive load balancing algorithms using symmetric broadcast networks: Performance study on an IBM SP2", *Proc. 26th International Conference on Parallel Processing* (1997), pp. 360–367.

[7] S.K. Das, D.J. Harvey, and R. Biswas, "Parallel Processing of Adaptive Meshes with Load Balancing," *Proc. 27th International Conference on Parallel Processing* (1998), to appear.

[8] S.K. Das and S.K. Prasad, "Implementing task ready queues in a multiprocessing environment," *Proc. International Conference on Parallel Computing* (1990), pp. 132–140.

[9] T. Eicken, D.E. Culler, S.C. Goldstein, and K.E. Schauser, "Active messages: A mechanism for integrated communication and computation," *Proc. 19th International Symposium on Computer Architecture* (1992).

[10] C. Farhat, "A simple and efficient automatic FEM domain decomposer," *Computers and Structures*, 28 (1988), pp. 579–602.

[11] J.E. Flaherty, R.M. Loy, C. Ozturan, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. Ziantz, "Parallel structures and dynamic load balancing for adaptive finite element computation," *Applied Numerical Mathematics*, 26 (1998), pp. 241–263.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam, *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Network Parallel Computing*, MIT Press, 1994.

[13] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," Technical Report SABD83-1391M, Sandia National Laboratories, Albuquerque, 1993.

[14] G. Horton, "A multi-level diffusion method for dynamic load balancing", *Parallel Computing*, 19 (1993), pp. 209–229.

[15] http://www.globus.org/

[16] http://science.nas.nasa.gov/Groups/Tools/IPG/

[17] http://www.capsl.udel.edu/EARTH/

[18] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," Technical Report 95-035, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, 1995.

[19] G. Karypis and V. Kumar, "Parallel multilevel K-way partitioning scheme for irregular graphs," Technical Report 96-036, Dept. of Computer Science, Univ. of Minnesota, Minneapolis, 1996.

[20] B.W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs", *Bell Systems Technical Journal*, 49 (1970), pp. 291–307.

[21] S. Khuri and A. Baterekh, "Genetic algorithms and discrete optimization," *Methods of Operations Research*, 64 (1991), pp. 133–142.

[22] S. Kirkpatrick, C.D. Gelatt Jr., and M.P. Vechhi, "Optimization by simulated annealing," *Science*, 220 (1983), pp. 671–680.

[23] G.A. Kohring, "Dynamic load balancing for parallelized particle simulations on MIMD computers," *Parallel Computing*, 21(1995), pp. 683–693.

[24] E. Leiss and H. Reddy, "Distributed load balancing: Design and performance analysis, *W.M. Keck Research Computation Laboratory*, 5 (1989), pp. 205–270.

[25] Message Passing Interface Forum, MPI: Message-Passing Interface Standard, Version 2, Technical Report, Univ. of Tennessee, Knoxville, 1997.

[26] B. Nour-Omid, A. Raefsky, and G. Lyzenga, "Solving finite element equations on concurrent computers," *Parallel Computations and their Impact on Mechanics* 1986, pp. 209.

[27] L. Oliker and R. Biswas, "PLUM: Parallel load balancing for adaptive unstructured meshes," Technical Report NAS-97-020, NASA Ames Research Center, Moffett Field, 1997.

[28] L. Oliker, R. Biswas, and H.N. Gabow, "Performance Analysis and Portability of the PLUM Load Balancing System," *Proceedings of Euro-Par'98 Parallel Processing*, Springer-Verlag, LNCS, to appear.

[29] L. Oliker, R. Biswas, and R.C. Strawn, "Parallel implementation of an adaptive scheme for 3d unstructured grids on the SP2," *Parallel Algorithms for Irregularly Structured Problems*, Springer-Verlag, LNCS 1117 (1996), pp. 35–47.

[30] H.D. Simon, "Partitioning of unstructured problems for parallel processing," *Computing Systems in Engineering*, 2 (1991), pp. 135–148.

[31] A. Sohn, Y. Kodama, J. Ku, M. Sato, H. Sakane, H. Yamana, S. Sakai, Y. Yamaguchi, "Fine-Grain Multithreading with the EM-X Multiprocessor," *Proc. 9th Symposium on Parallel Algorithms and Architectures* (1997), pp. 189–198.

[32] R. Van Driessche and D. Roose, "Load balancing computational fluid dynamics calculations on unstructured grids," *Parallel Computing in CFD*, AGARD-R-807 (1995), pp. 2.1–2.26.

[33] A. Vidwans, Y. Kallinderis, and V. Venkatakrishnan, "Parallel dynamic load balancing algorithm for three-dimensional adaptive unstructured grids," *AIAA Journal*, 32 (1994), pp. 495–505.

[34] C. Walshaw, M. Cross, and M.G. Everett, "Parallel dynamic graph partitioning for adaptive unstructured meshes," *Journal of Parallel and Distributed Computing*, 47 (1997), pp. 102–108.